Argonne NATIONAL LABORATORY

Advanced CODES/ROSS Usage and Strategies

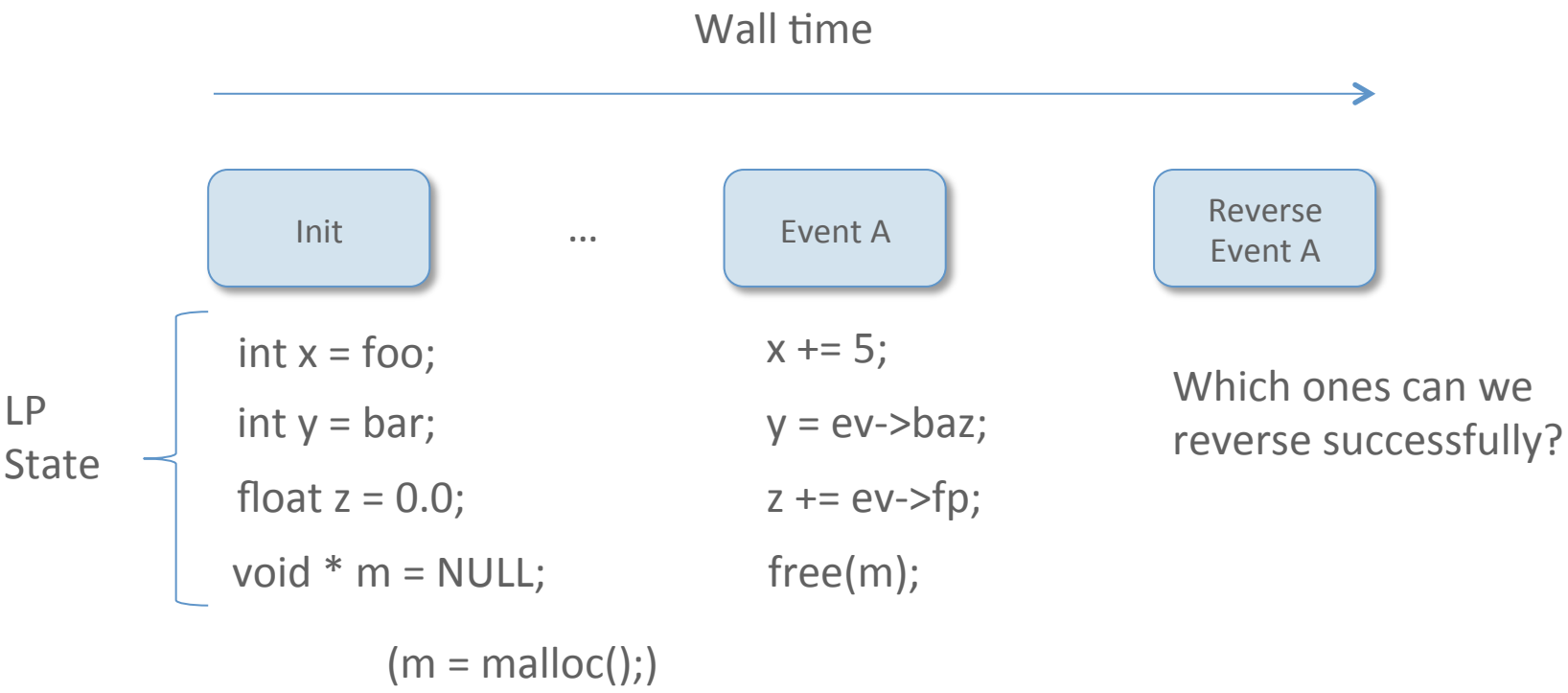
John Jenkins, ANL

U.S. DEPARTMENT OF ENERGY

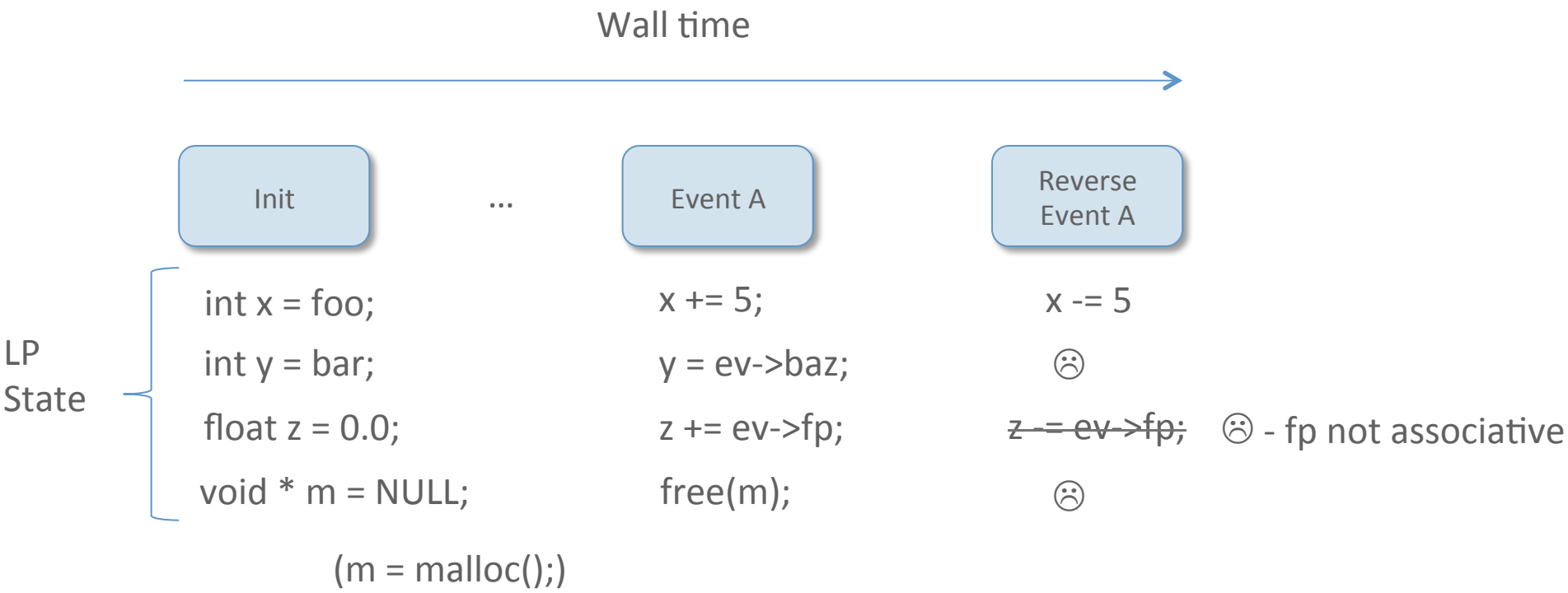
Outline

- Sample pain points of using an optimistic PDES in general, ROSS/CODES specifically
- Identify mitigation strategies
- Profit

Optimistic mode is annoying (state mutation)



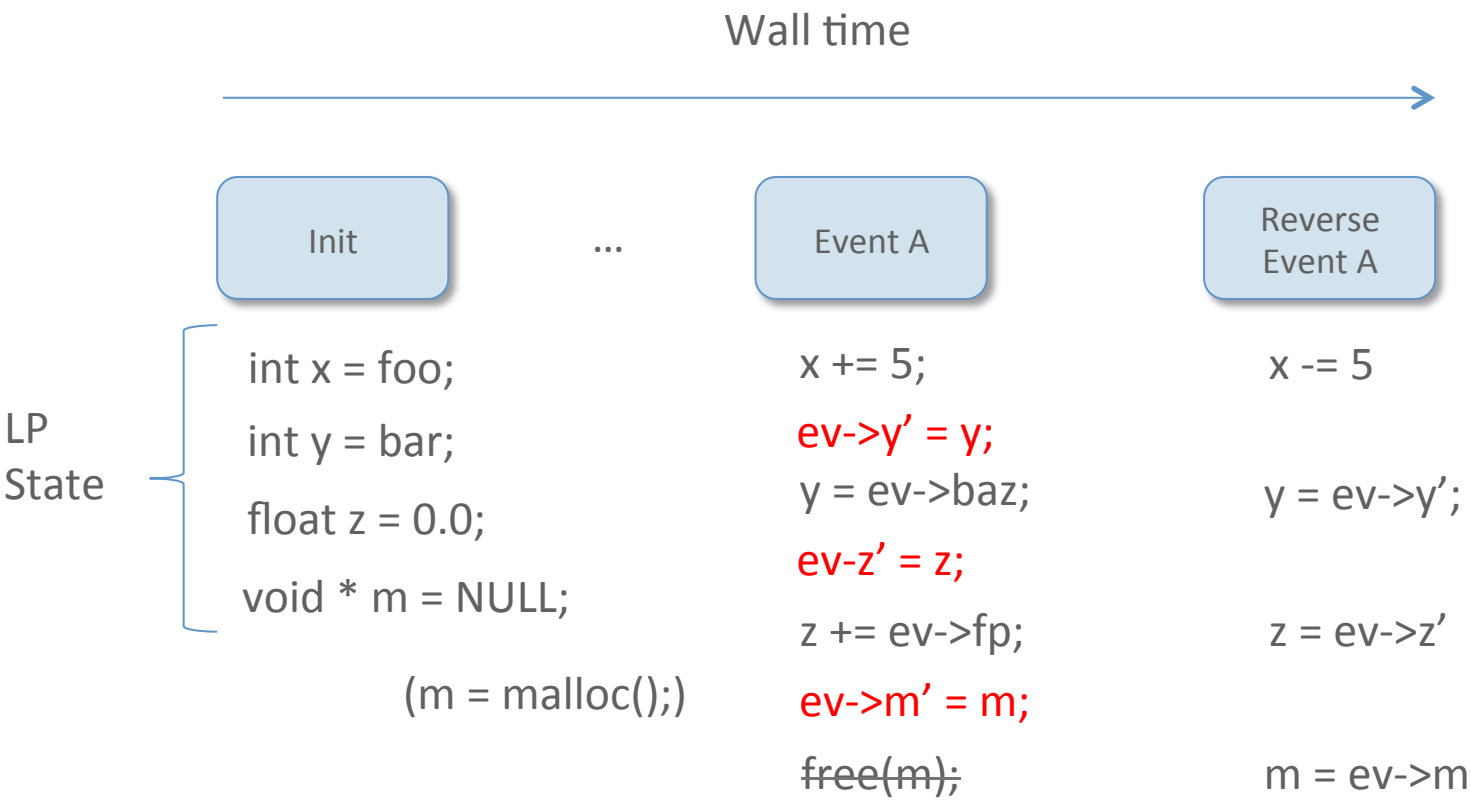
Optimistic mode is annoying (state mutation)



Optimistic mode is annoying (state mutation)

Solution – save state for **destructive** operations

- destructive – FP operations, assignment, free, etc. (not even thinking about IO…)
- in the event is a good place to do this
  - same event mem used for forward and reverse handler



Everything good??

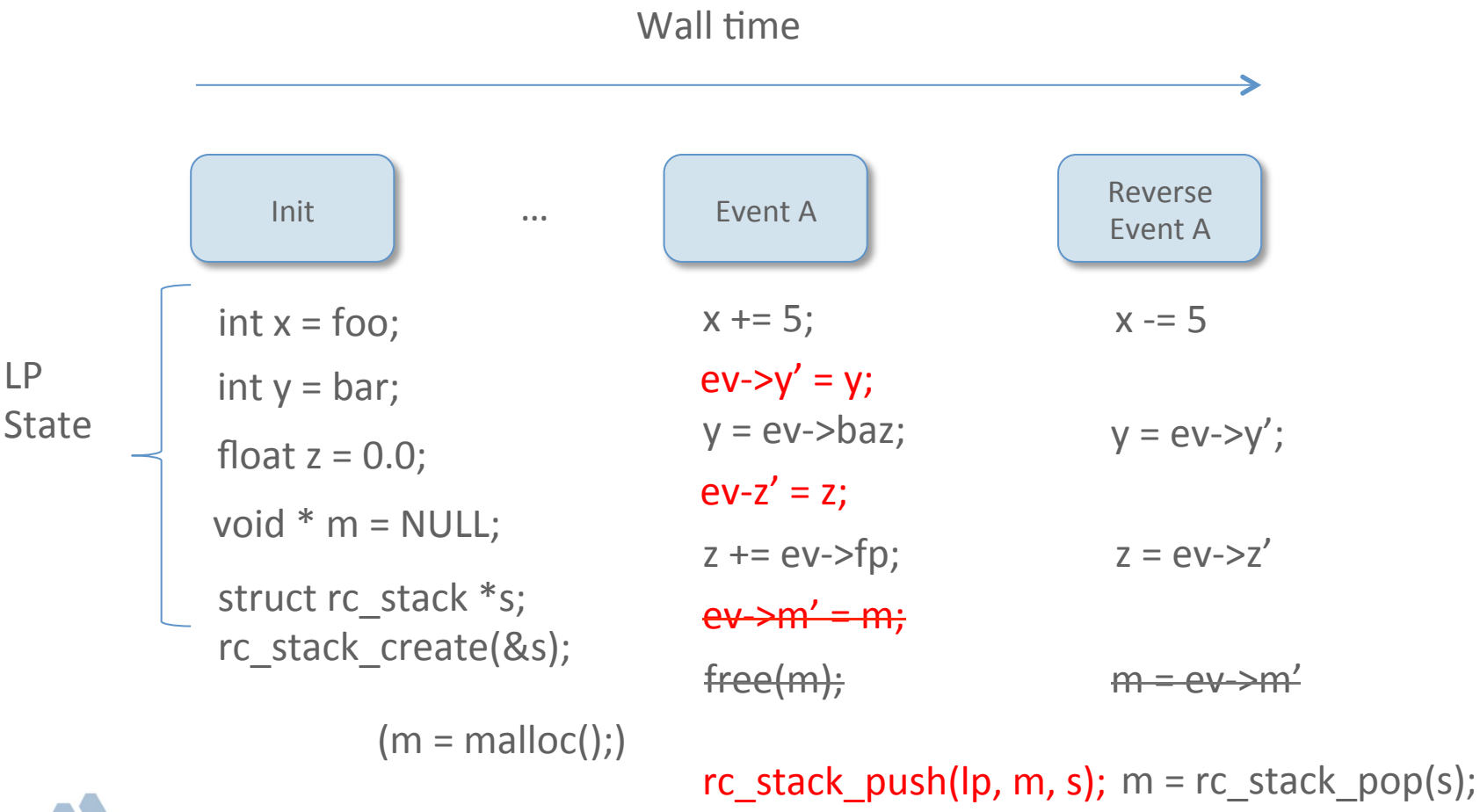
Optimistic mode is annoying (state mutation)

Problem – memory management (memory leaks in handling of m)

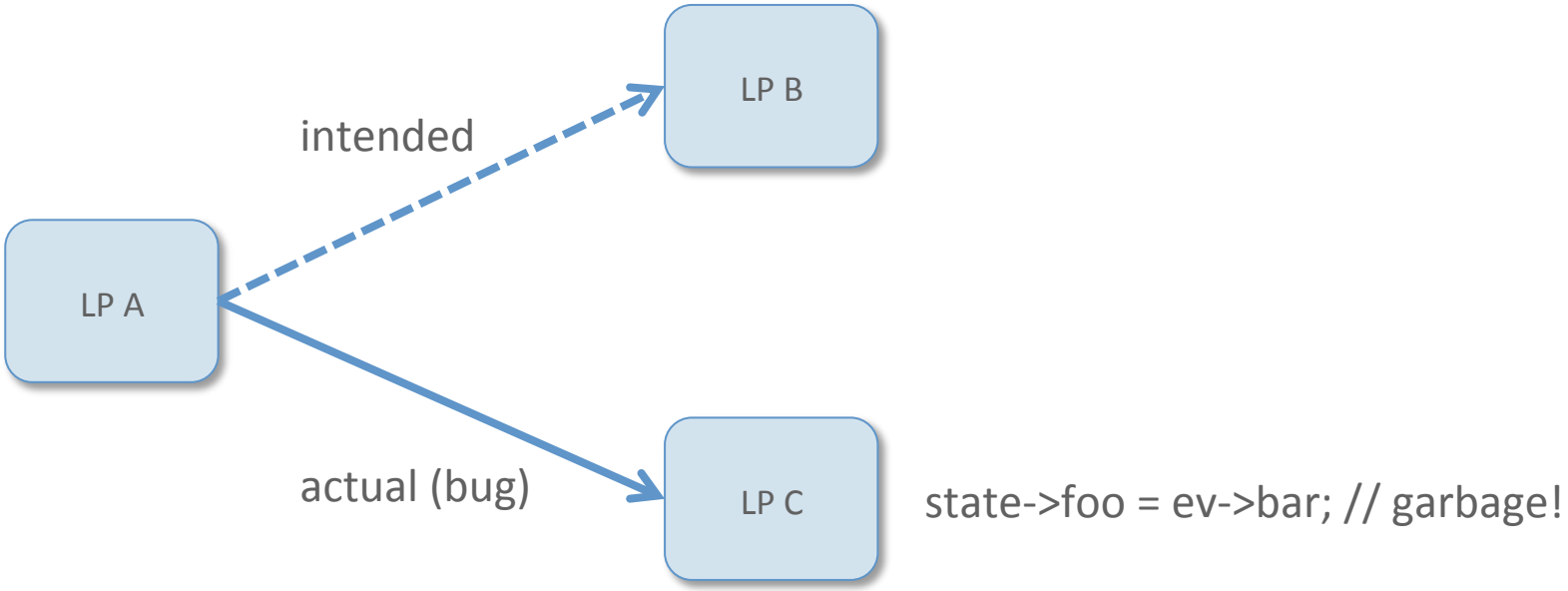
- need to keep around memory, but don't know when to free it ☹

Solution – CODES! (codes/rc-stack.h)

- use a stack data structure of pointers, garbage-collect based on GVT



Optimistic mode is still annoying (control flow)

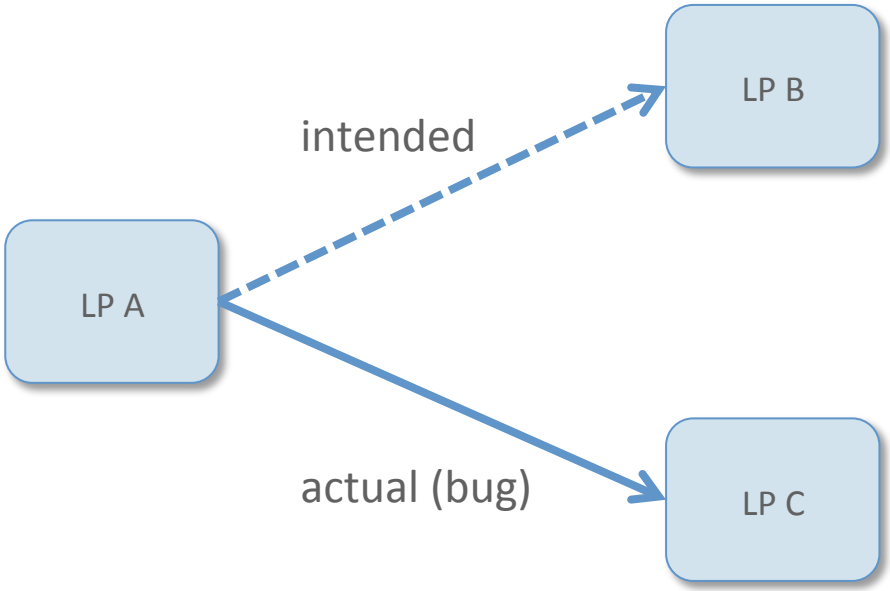


ROSS does not help you with this – can't "type check" your messages

Optimistic mode is still annoying (control flow)

Solution: BYO consistent event structuring

- CODES can help (codes/lp-msg.h)




```
struct event_b {
        msg_header h;
        …
};

 struct event_c {
        msg_header h;
        …
 };
```

```
assert(lp_type_magic == ev->h.magic);
```

```
struct msg_header {
   tw_lpid src;
   int event_type;
   int magic; // magic number for recipient type
};
```

Optimistic mode is still annoying (etc)

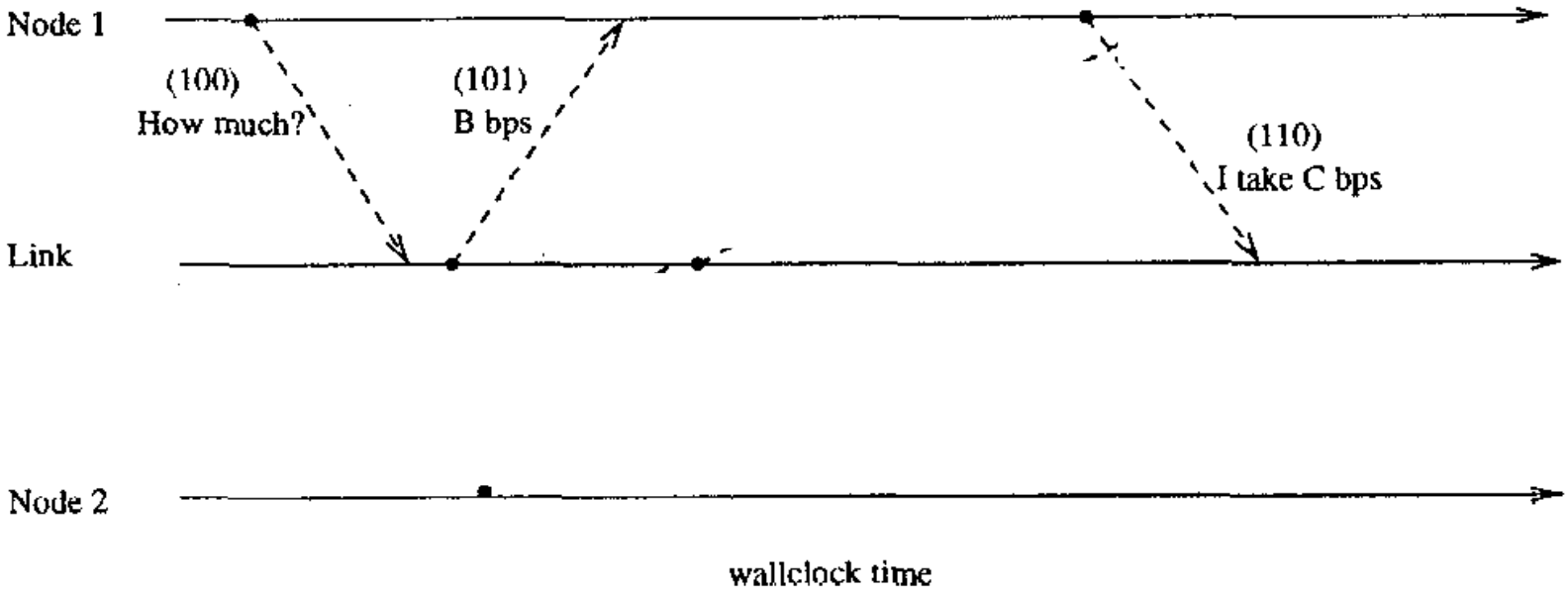
- Misc. recommendations
  - Use bitfields for complicated conditionals (tw_bf, available with every event)
  - Structure code to minimize mixing of state mutation and control flow based on mutated state
  - Very complicated control flows -- while (…) { if (…) { mutate_state } }
    - refactor into multiple passes
    - refactor into multiple events, using self-events for control flow
  - Use OPTIMISTIC_DEBUG mode (--sync=4) to debug general reverse computation behavior
    - Runs forward until out of event memory, then reverses to the beginning
  - Discuss on the mailing list (codes-ross-users@lists.mcs.anl.gov) ☺

CODES/ROSS helpers for optimistic mode

- rc_stack_* (codes/rc_stack.h)
  - Lazy free list, allows for (user-driven) garbage collection based on GVT
- lp_io_* (codes/lp-io.h)
  - Reverse computation aware file output for modest data sizes
  - Similar to tw_printf, but uses MPI collectives at end of sim to combine output
  - lp-io support in model-net, local storage models (“category” function param)
- msg_header (codes/lp-msg.h)
  - Commonly used event variables (src LP-ID, event type marker, lp type “magic” number)
    - CODES convention – magic = hash(lp_name);
- tw_output (ROSS/core/ross-extern.h)
  - Optimistic-aware printf (prints on GVT)
- Optimistic debug mode (--sync=4) – use it!!!

Optimistic mode is hard!

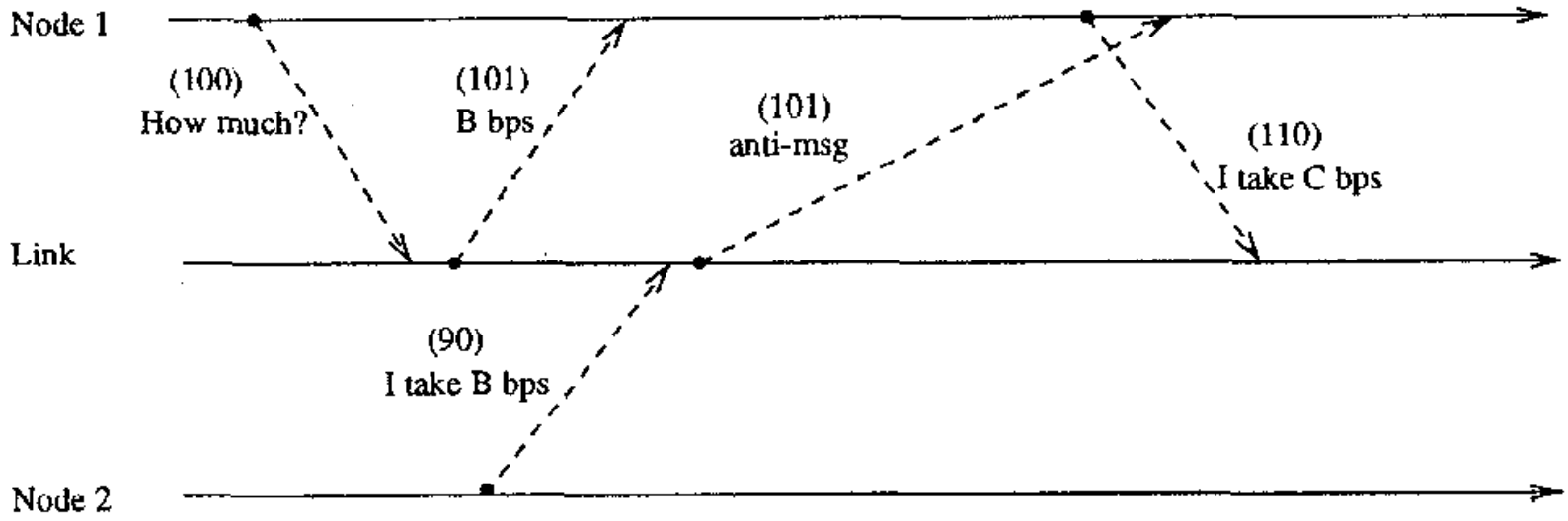
- Optimistic concurrency comes at a price
  - Reverse computation is programmer-provided
  - Emergent multi-event effects may break model assumptions



From D. M. Nicol and X. Liu, "The dark side of risk (what your mother never told you about Time Warp)," in , 11th Workshop on Parallel and Distributed Simulation, 1997., Proceedings, 1997, pp. 188–195.

Optimistic mode is hard!

- Model assumption: link LP doesn't broadcast more available b/w than it can allocate.
- Message at simulation time 110 exists outside of reality!
  - Node sends message thinking everything is OK
    - Based on an "alternate timeline"
  - Link receives message that is inconsistent with it's view of the world
    - But message appears legitimate!
  - What if you freed request memory, shrunk your array size, done most anything in C?
- Optimistic debug mode doesn't help with this!



Coping strategies

- Defensive programming!
  - Aggressively check model assumptions for unexpected behavior
    - Especially for complicated data structure handling
- Use the self-suspend technique

Self-suspend

```
struct lp_state {
        …
        int suspend; // init to 0
}
```

```
void event(lp_state *s, …) {
        if (suspend) { // ignore event
                // can see multiple events
                // before rollback
                suspend++;
                return;
        }

        …
        if (broken_model_assumption) {
                suspend = 1;
                // use codes/lp-io.h for
                // optimistic-aware output
                lp_io_write("error: …");
                return;
                tw_error(…);
        }
}
```

```
void revent(lp_state *s, …) {
        // do nothing for ignored events,
        // *until* we're back at the originating event
        if (suspend && --suspend) {
                return;
        }

        // reverse event code
}
```

Idea - restrict the set of invalid states your LP sees
- don't spend time crunching numbers that will get rolled back anyways

Wrapping up

- For more tips:
  - check out the CODES best practices document (doc/codes-best-practices.tex – use the makefile to build the pdf)
  - check out the ROSS wiki (https://github.com/carothersc/ROSS/wiki)
- Lots more I didn't cover here:
  - Encapsulation of message types between different LPs
  - Sane, generic interfaces into LPs
  - More coding-specific tips
    - Modelnet, other codes models
    - Configuration strategies
    - Optimizing models
  - Let's discuss these during the hackathon!